Department of Computer Science
Columbia University
Technical Report Series: CUCS-002-09

450 Computer Science Building
1214 Amsterdam Ave., MC 0401
New York City, NY 10027-7003
http://www.cs.columbia.edu

Rapid Parallelization By Collaboration

Simha Sethumadhavan
Computer Architecture Lab
simha@cs.columbia.edu

Gail. E. Kaiser
Programming Systems Lab
kaiser@cs.columbia.edu

Published January 1 2009

Abstract

The widespread adoption of Chip Multiprocessors has renewed the emphasis on the use of parallelism to improve performance. The present and growing diversity in hardware architectures and software environments, however, continues to pose difficulties in the effective use of parallelism thus delaying a quick and smooth transition to the concurrency era.

In this document, we describe the research being conducted at Columbia University on a system called **COMPASS** that aims to simplify this transition by providing advice to programmers considering parallelizing their code. The advice proffered to the programmer is based on the wisdom collected from programmers who have already parallelized some code. The utility of **COMPASS** rests, not only on its ability to collect the wisdom unintrusively but also on its ablility to *automatically* seek, find and synthesize this wisdom into advice that is tailored to the code the user is considering parallelizing and to the environment in which the optimized program will execute in.

COMPASS provides a platform and an extensible framework for sharing human expertise about code parallelization – widely, and on diverse hardware and software. By leveraging the "wisdom of crowds" model [78] which has been conjunctured to scale exponentially and which has successfully worked for wikis, **COMPASS** aims to enable *rapid* parallelization of code and thus continue to extend the benefits of Moore's law scaling to science and society.

ACM Keywords: **C.1.4:** Parallel architectures **D.3.4:** Software optimization **F.3.2:** Program analysis **H.3.4:** Recommender systems **I.2.6:** Knowledge Acquisition **K.4.3:** Computer-supported Collaborative Work

Prelude: The Story of Jo the Programmer

Jo the programmer is a smart, competent, hard working chief software engineer at a leading firm. In a recent company meeting, representatives from the company's primary hardware vendor informed management that all future hardware will be "multi-core" (e.g., 8 cores), and will soon be "many-core" (arbitrarily referring to more than 16) [7]. The manager of the hardware team that maintains the company's server farm is thrilled because of the power and space benefits [31]. But Jo is not; Jo realizes that the performance of the company's enterprise software will no longer automatically keep pace with increasing user demand just by buying higher clock frequency (faster) processors [2, 49, 56]. So Jo's staff now has to explicitly redesign the software to take advantage of the new parallelism.

Jo sets out on a mission to learn about parallelization options. After all, Jo's staff will soon be asking Jo for advice; some of them have been professionals for decades but without any parallelization experience, others are entry-level programmers who learned a little bit about "threads" in college. What Jo finds out is disconcerting. Through a few informed Internet searches, Jo discovers that numerous researchers have raised concerns over the issue and have proposed multi-year projects to invent new languages [13, 38, 42], new compiler techniques [55, 59, 73] and/or new hardware [15, 66, 68]. Jo also finds significant diversity in the numerous already available parallelization techniques, some of which involve special purpose processors such as for graphics [57], some are specific to embedded devices [35], and some applicable to multicore servers and desktops [62]. Jo feels lost because all Jo remembers from grad school were a few lectures on synchronization and a guest speaker about supercomputers for scientific computing (not particularly relevant to Jo's company). Jo considers the available options and risks: (1) train the entire software development department to use some parallel programming language and translate all legacy programs to that language; (2) wait a few years for compiler parallelization techniques to be available for their current language; or (3) live with user complaints about slower enterprise applications. Jo thinks the first option could work but does not know what language to adopt; researchers have clearly pointed out problems with current languages, but their new languages don't yet have good tool support. Jo is back to square one.

The solution we present here is designed to help people like Jo and Jo's staff to break this conundrum and transition smoothly to the concurrency era.

1 Introduction

Inspired by a growing set of techniques that leverage collective human intelligence to solve difficult problems in computer science such as image recognition [80], we describe a collaborative human-based computation system that attacks another difficult problem: parallelizing and optimizing sequential programs for highly parallel Chip MultiProcessor (CMP) architectures. Our system, called **COMPASS** (A Community-driven Parallelization Advisor for Sequential Software), proffers advice to programmers based on information collected from observing a community of programmers parallelize their code. The utility of **COMPASS** rests in part on the premise that the growing popularity of CMP systems will encourage expert programmers to parallelize some of the existing sequential software, and **COMPASS** can quickly deploy capabilities to capture their wisdom (including any gained through trial and error intermediate steps) for the community.

COMPASS is capable of providing a wide range of advice for improving performance by using different granularities of parallelism and its exact use depends on the nature and number of users in the system. It is not unreasonable to imagine that **COMPASS** will be able to provide advice for a large class of present and upcoming CMP systems including: peephole Data Level Parallelism optimizations such as using SIMD kernels for inner loops, splitting loop iterations into threads using OpenMP, replacement and threading of large chunks of serial code with vendor provided optimized libraries (such as Nvidia CUDA), template for stream dataflow graphs or thread and lock creation for irregular applications.

COMPASS observes expert programmers (henceforth called gurus) parallelize their sequential code using one or more of the numerous techniques for parallelization, records their code changes, summarizes this information and stores it in a centralized Internet-accessible database. When an inexperienced new user (henceforth called a learner) wants to parallelize his/her code, the system first identifies the regions of code most warranting performance improvement (identified by profiling typical executions), and then that are most amenable to parallelization (by consulting the database of previously parallelized code). **COMPASS** then presents a stylized template, or "sketch", that can be used as a starting point for parallelization by the learner. The learner may then provide feedback to the system on the usefulness of the advice. To effectively provide these capabilities, **COMPASS** introduces a new program abstraction called a *Segment Dependence Graph* (SDG), which represents program segments in an easily comparable and platform-independent way that removes most problems with data structure aliasing. A SDG is created using a novel lightweight program analysis technique that combines both static and dynamic measurements of program dependencies.

COMPASS is a significant improvement over the state of the art, where the learner has to "pull" parallelization advice from books, class notes, and/or Internet tutorial examples. Such a process can be time consuming and error prone. **COMPASS** on the other hand is a "push" based system where advice is proactively proferred to the learner with very little involvement of the learner. In addition, unlike tutorials, the advice is customized to the learner's coding problem at hand. Further, unlike traditional hardware, compiler, language, or hybrid approaches for parallelization which have long incubation times before the end users can benefit, typically years to decades, **COMPASS** can enable rapid parallelization of sequential codes because the usefulness of knowledge networks, such as **COMPASS**, scales exponentially as the number of users in the system increases [64].

2 Related Work

To the best of our knowledge, we are the first to propose leveraging collective human wisdom to propagate "best practices" about how to parallelize code. This is a new way of thinking about performance engineering enabled by the connectedness brought about the Internet and the wide availability of parallel machines.

The problem of parallelizing programs for high performance is, however, well-documented (e.g., papers from 1950s/60s [17, 27, 44]), perhaps one of the fundamental problems of computer science, and a rich set of techniques has been proposed to tackle this challenge. These techniques fall into one of five categories: pure hardware solutions, pure compiler optimizations, pure language solutions, hardware-compiler co-operative techniques or compiler-language solutions, and have had qualified successes but overall the problem still remains challenging when compared to the achievable potential [36, 63].

Hardware-only solutions improve performance by executing independent instructions in parallel, and possibly out-of-order, from a pool of sequentially fetched instructions [39]. The larger the pool of instructions, the higher the likelihood of finding independent instructions, and the greater chance of the performance improvements. However, traditional hardware-only techniques currently cannot increase the pool of instructions beyond the low hundreds [71] because of heat dissipation and slow operation speed [2]. Pure compiler solutions for automatic parallelization, the "holy grail" of compilers, use static analysis to recognize and execute independent regions of code in parallel but are limited due to the challenges in scaling pointer analysis, loss of information in intermediate representations, and static nature of optimizations [19, 69]. Pure language solutions for high performance, in an attempt to make the code amenable to compiler (pointer) analysis for parallelism, limit practical language features – which can hamper their use. For example, Fortran [12] restricts the use of pointers, and functional languages [43] have Write-once semantics. In other cases, languages require a high degree of user intervention for several parallelization tasks which can diminish productivity. Although co-operative hardware/software techniques hold significant promise and can be used to extract parallelism from a pool of thousands of instructions [67], these systems require sophisticated compiler techniques that are still being developed. Language-compiler solutions, including annotation languages such as OpenMP [16], are used by the programmer to communicate hints to the compiler. While promising, they have not been widely used beyond the High Performance Community. Researchers are working to improve upon these traditional approaches, but **COMPASS** takes a fundamentally different tack, instead aiming to parallelize code by collecting parallelization knowledge from a massive number of programmers, some of whom may be researchers. **COMPASS** is thus complementary to those proposals. For example, new languages such as Atomos [13] and Streamware [38] aim to simplify particular aspects of parallel programming, like synchronization or orchestration between tasks, while **COMPASS** can help a learner in the first step of decomposing a sequential program into tasks in addition to the above tasks.

Our solution is also related to work on fields of Human Computation Systems (HCS) and Knowledge Engineering and Management (KEM). **COMPASS** is similar to HCS applications such as games with a purpose, or prediction markets where human behavior is captured by computers to solve problems. **COMPASS** is also similar to KEM systems, such as wikis, because it stores and manages user knowledge. However, for both these areas we are not aware of analogous applications, systems and techniques for handling code.

3 Collaborative Code Mining Systems Desiderata

COMPASS is one instantiation of a large (and new) class of collaborative systems that promotes "best practices" for a specified task based on data collected from mining codebases for similarity and then synthesizing the data to provide customized recommendations with little human intervention. Such collaborative systems, including **COMPASS**, should have five characteristics to be successful – Proactivity, Specificity, Universality, Scalability and Anonymity. We discuss these in the context of **COMPASS** and parallelization as they are most relevant, but adaptation of these criteria to other systems that rely on code mining are straightforward.

§ **Proactivity** The state of the art for sharing expertise about parallelization is primarily through classroom teaching [72], books [14, 23, 54], Internet tutorials [52, 53], mailing lists and blogs [40, 41]. These approaches can be characterized as "pull" systems, since the learner has to seek out and pull the advice (s)he needs. While traditional pull systems are valuable, a collaborative system that aims to do better must rectify a very important drawback, *Vis-á-vis*, the "know the unknown" problem. To find a suitable optimization using these systems, the learner typically has to know (the name) of the optimization that the learner does not know! To workaround this chicken-and-egg situation, the learner might perform a series of trial-and-error search engine queries to find the correct optimizations – which can be time consuming and frustrating. If the learner turns to a mailing list for help, long turnaround times may result from the typical translation of an optimization problem from computer language to natural language (at the requesting learner's side), then from that natural language to an optimization in computer language back to natural language (at the responding guru's side), and finally back to computer language from that natural language (returning to the learner's side). To address this drawback, a collaborative system must function under a "push" model where the system proactively fetches useful advice with minimal learner intervention, in a form reasonably close to the original computer language, based on some analysis of the code to be parallelized.

§ **Specificity** Another serious drawback of traditional pull systems is the lack of advice specificity. For instance, it is unlikely that the learner will find a tutorial working through the exact code sample that (s)he is trying to parallelize. Instead, the learner might try to adapt a tutorial example to his/her code – which is error prone. If a mailing list is used to request specific help, some vital information may be lost in the natural language translation in either direction between the guru and the learner. (We assume few such gurus will have sufficient free time to perform the parallelization directly on learner-provided code, and anyone so kind would soon be inundated.) A collaborative parallelization system has potential to workaround this problem, providing very specific and accurate automated advice based on optimizations available in before and after form on real-world code instead of toy tutorial examples. To realize this potential, however, the collaborative system should provide advice in a format that reduces information loss and errors, and thus improves overall productivity. Furthermore, when more than one approach is known, the system must filter and rank the advice based on some heuristic(s) and serve the selected advice in a prioritized manner.

§ **Universality** A collaborative parallelization system should be able to collect and report knowledge from a wide variety of sources and support most techniques equally well. To understand the range of optimizations such a collaborative system must handle, consider an expert programmer's parallelization "grab bag." The grab bag is likely to contain optimizations to exploit concurrency (a) at different granularities:

instruction-level [10, 39], data-level [1, 65] and task-level [4]; (b) for different memory models: shared memory or message passing [24]; (c) under different assumptions about system software and hardware capabilities: type of architecture [6, 34, 76, 79], type of compiler, type of hardware acceleration units (e.g., physics [22], network [18] or graphics [61] processing unit), type of special purpose libaries (e.g., LAPACK [11], CUDA [57]), etc.; or (d) for different application domains: scientific [23], embedded [28] or server [77], to name a few. To accommodate all of these techniques in its knowledge base, the collaborative system should use an internal representation that is universal, i.e., portable across target platforms and with enough semantic power to capture the entire range of parallel optimizations.

§ **Anonymity** Some users of the collaborative system may not wish to share exact information about their code (e.g., employees of commercial organizations). In such cases, the collaborative system must be able to forego extracting that user's optimizations (in a guru's role) as well as return advice based only on the data a user is willing to share (in a learner's role), with the implicit understanding that the quality of supplied advice may not be the best available in such cases. Further, the database should be effectively stateless if the user so desires, in the sense of treating each request-response and each optimization submission (if any) entirely independently. Further, organizations should be able to host their own private instance of the collaborative system for internal use if they so desire.

§ **Scalability** To be genuinely useful to the large community we envision, a collaborative parallelization system must provide an Internet-scale database to collect, aggregate and disseminate knowledge about optimizations. Storing this knowledge in some repository providing a database interface via programmatic Internet access is critical not only for archiving parallelization optimizations in an easy-to-use, shareable format, but more importantly to allow users to build their own new systems that provide not-yet-known enhanced functionality utilizing the data. In previous parallel computing revolutions, most notably the high performance computing-inspired revolution of the 1980s and 1990s, valuable knowledge on parallel optimizations was archived in journal articles (in the best case) or limited to only the few individuals working on a particular system. Inter-networking was concentrated in small pockets and could not be leveraged for dissemination on the scale possible today; collaborative parallelization systems should take advantange of open Internet technology to promote maximum dissemination of ideas and innovations discovered during the current multicore-inspired parallel computing revolution. After a period of use, the collaborative system could support tens of thousands of programmers and projects with tens of millions of lines of code. A highly scalable database schema and corresponding interface is required for storing and retrieving these optimizations.

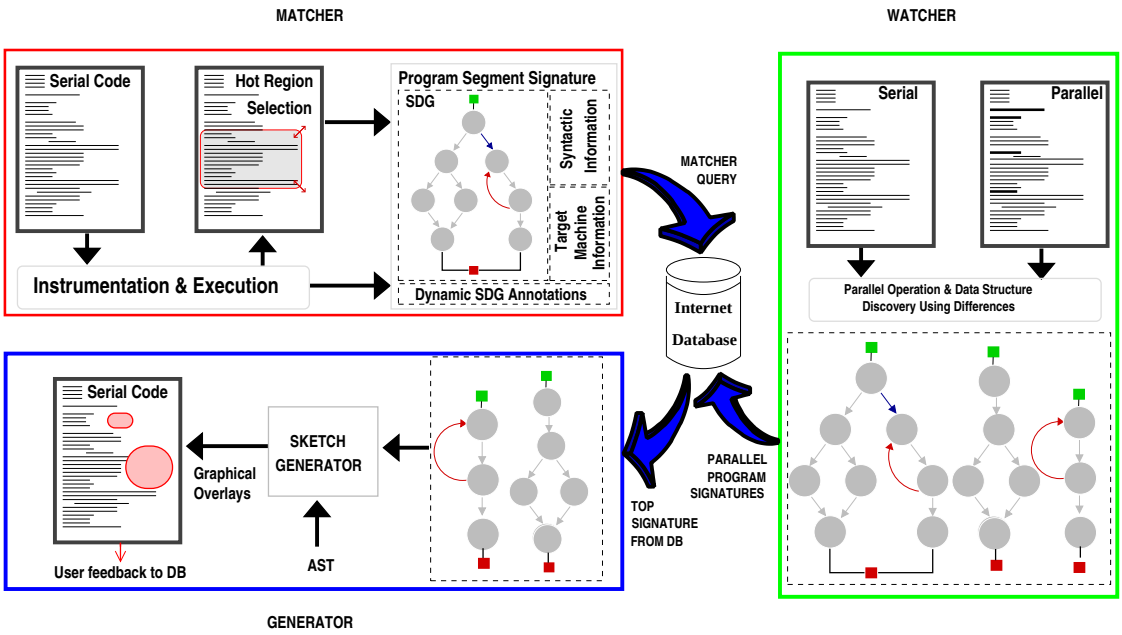

Figure 1: System architecture of the proposed **COMPASS** system.

4 COMPASS: System Architecture and Use Cases

COMPASS targets two main groups of human actors: the gurus, who are experienced with parallel optimizations, and the learners, who are attempting to optimize their code for multi-/many-core machines, perhaps for the first time. Its a continuum: The same individual may operate as both guru and learner, presumably in different contexts and/or as expertise builds over time. We assume some initial set of (relative) gurus, rather than all learners. The purpose of **COMPASS** is to provide a framework and a set of mechanisms that makes it simple for the gurus to communicate their wisdom to the learners. One key aspect of the system is that the wisdom to be offered to learners is gathered from *multiple* gurus, some of which may not trustworthy or optimal – then mined, customized and served with very little effort by the learners.

The architecture of **COMPASS** is shown in Figure 1. The system has four modules that combine to provide the proposed functionality – the watcher, the data store, the matcher, and the generator. The watcher observes how the gurus parallelize their code by tracking the differences in the source code before and after an optimization. When an optimization is complete, the watcher snips out the code regions corresponding to the optimization, and summarizes the before and after versions in the form of unique identifiers, called signatures, and deposits them in the data store. The data store holds the before and after signatures as $< key, value >$ pairs (with the before version acting as the key and the after version serving as a value – which could in principle be reversed to de-parallelize if warranted). When a learner wants to parallelize some code, the matcher module prepares signatures of regions of code that are considered critical for parallelization (e.g., via hotspot profiling), and sends them to the data store. The data store runs a query with the matcher signature as a key, ranks the results and returns the top parallelized signature(s) to the generator. The generator then prepares a "sketch" of the code corresponding to the parallelized version, presented as a graphical overlay that can be accepted as is, or modified or rejected by the learner. The learner may optionally provide feedback to the data store on the usefulness of the sketch.

4.1 COMPASS Use Cases

We envision **COMPASS** being useful in a wide range of optimization scenarios, only some of which we discuss below. These scenarios are chosen to highlight parallelization opportunities that **COMPASS** is likely to be able to exploit whereas typical optimizers – compilers and hardware – do not or, in some cases, can not exploit. Further, these optimizations concern code snippets that occur quite often. The optimizations cover both control and data changes, and consider different program granularities (peephole to global). We believe that over time, the number and sophistication of optimizations that **COMPASS** can support will grow, as more gurus join the system, and will ultimately be limited only by human ingenuity (assuming scalability).

§ **Peephole Optimizations** The code snippet listed in Figure 2 is part of the Inverse Discrete Cosine Transform algorithm which is widely used in popular image and video applications such as libJPEG, FFmpeg and H.263 decoder. This code was tested with GCC4.3, MS VC++ (2008) and Intel 10.1 CC compilers for auto-vectorization – a common peephole data parallel optimization. All these compilers fail to auto-vectorize the code listed on the LHS; probably because of data dependent conditionals (the source code for some these compilers are not available to us). The code on the RHS is the hand optimized version using x86 SSE instructions and results in a speed up of 2x to 3x when tested with GCC, with optimization flags. **COMPASS** can recognize this optimization and disseminate it to the community, even though only one of the gurus, such as a student in a university class on parallel programming or parallel computer architecture had manually optimized this code during database corpus pre-population. To preserve maintainability, the advice from **COMPASS** is always enclosed in `#ifdef` as shown in the RHS of the figure.

§ **Loop Parallelization** A commonly used optimization to speed up programs is to execute *independent* iterations of the loop in parallel on different cores. The efficacy of the optimization depends on the ability

```
/*Before*/
void foo(unsigned char output[64], int Yc[64], int S_BITS){




 for (int l = 0; l < 8; l++) {
     for (int k = 0; k < 8; k++) {
         int temp = Yc[k] + ( 1 << (S_BITS + 2 ));
         temp = temp - (Yc[k] < 0);
         temp = temp >> (S_BITS + 3);
         int r = 128 + temp;
         r = r > 0 ? (r < 255 ? r : 255) : 0;
         output[k*8+ l] = r;
     }
 }




}
```

```
/* After */
void foo(unsigned char output[64], int Yc[64], int S_BITS) {
    #ifndef COMPASS
      // original code goes here
    #else
       __m128i XMM1, XMM2, XMM3, XMM4;
       __m128i *xmm1 = (__m128i*)Yc;
       __m128i XMM5 = _mm_cvtsi32_si128(S_BITS + 3) ;
       __m128i const_128 = _mm_set1_epi32(128);
       __m128i zero = _mm_setzero_si128();
       XMM2 = _mm_set1_epi32(S_BITS + 2);
       for (int l = 0; l < 8; l++) {
           XMM1 = _mm_loadu_si128(xmm1++);
           XMM3 = _mm_add_epi32(XMM1, XMM2);
           XMM1 = _mm_cmplt_epi32(XMM1, zero);
           XMM1 = _mm_srli_epi32(XMM1, 31);
           XMM3 = _mm_sub_epi32(XMM3, XMM1);
           XMM3 = _mm_srl_epi32(XMM3, XMM5);
           XMM3 = _mm_add_epi32(XMM3, const_128);

           XMM1 = _mm_loadu_si128(xmm1++);
           XMM4 = _mm_add_epi32(XMM1, XMM2);
           XMM1 = _mm_cmplt_epi32(XMM1, zero);
           XMM1 = _mm_srli_epi32(XMM1, 31);
           XMM4 = _mm_sub_epi32(XMM4, XMM1);
           XMM4 = _mm_srl_epi32(XMM4, XMM5);
           XMM4 = _mm_add_epi32(XMM4, const_128);

           XMM3 = _mm_packs_epi32(XMM3, XMM4);
           XMM3 = _mm_packus_epi16(XMM3, XMM3);
           _mm_storel_epi64
                ((__m128i*)(output+8*l), XMM3);
       }
    #endif  }
```

Figure 2: Example **COMPASS** Peephole Optimization using SSE x86 instructions.

Figure 3: Example **COMPASS** Proceduralization using a library call from the Intel IPP®library [76]

```
void RGBToYCbCr_JPEG_8u_C3P3R(Ipp8u* pSrcRGB,
        int srcStep, Ipp8u* pDstYCbCr[3],
        int dstStep, IppiSize roiSize)
{
    for(int i = 0; i < roiSize.height; i++)

        for(int j=0; j < roiSize.width; j++)
        {
            int index = i * roiSize.width + j * 3;
            unsigned char R = pSrcRGB[ index];
            unsigned char G = pSrcRGB[ index + 1];
            unsigned char B = pSrcRGB[ index + 2];

            pDstYCbCr[0][i * roiSize.width + j]
             = 0.299*R + 0.587*G + 0.114*B;
            pDstYCbCr[1][i * roiSize.width + j]
             = -0.16874*R - 0.33126*G + 0.5*B + 128;
            pDstYCbCr[2][i * roiSize.width + j]
             = 0.5*R - 0.41869*G - 0.08131*B + 128;
        }
} /*Before */
```

```
void RGBToYCbCr_JPEG_8u_C3P3R(Ipp8u* pSrcRGB,
        int srcStep, Ipp8u* pDstYCbCr[3],
        int dstStep, IppiSize roiSize)
{
    #ifdef COMPASS
     ippiRGBToYCbCr_JPEG_8u_C3P3R(pSrcRGB,
     srcStep, pDstYCbCr, dstStep, roiSize);
    #else
       // original code
           #endif




   /* After */
```

of the compiler to determine independence of the loop iterations. Because of the limitations of pointer/alias analysis, modern production compilers often fail to indentify independence and are thus unable to parallelize such loops. To overcome this, annotation languages like OpenMP [16] are commonly used to help the compiler identify independent regions. **COMPASS** may be able to automatically determine these independent regions based on code similarity, and then suggest the correct OpenMP keywords to use.

§ **Proceduralization** When chip companies design new hardware features to improve performance, they typically also release APIs that can fully utilize the new hardware capabilities. Examples include the CUDA graphics library for effectively using the capabilities of NVIDIA Graphics Co-Processors [57] and Intel Performance Primitives [76] which contain specially optimized library routines for domain-specific operations such as JPEG image processing or video transcoding using the SSE instruction extensions. To improve performance using these libraries without **COMPASS**, *each* code owner has to know about the existence of the optimized API, know to use the API correctly, and then carefully adapt the code to invoke the API. On the other hand, with **COMPASS** advice, the developer is automatically informed of the changes and gets suggestions on code segments that can be proceduralized as soon as a few gurus have committed their corresponding changes. An example of such an replacement is shown in Figure 3. The LHS shows code listing routine from an open source library (libJPEG), been replaced on the RHS by an equivalent function call from the Intel Performance Primitives library.

§ **Locality Optimizations** The optimizations discussed so far apply to contiguous code regions (of increasing scope) and require only control changes. However, some parallelization optimizations require multiple changes in discontiguous regions. An example of such an optimization is one that improves the locality of a parallel program by changing the layout of the data structures in memory ([24], pp 145–148). There are two practical difficulties with locality optimizations: (1) the data structure occurence must be changed at all its locations in the code, and (2) it is difficult to predict whether the change will actually improve performance because these changes are very sensitive to machine configuration (e.g., L1/L2 cache size, line size, etc.). A combination of the above factors typically discourages programmers from trying such optimizations. **COMPASS** can mitigate some of these difficulties by distributing locality-related advice if and only if the gurus' configured target machines are equivalent to the learner's target. Data structure changes can also be identified easily by presenting the data store outputs as a code overlay.

5 Innovations to Support Collaborative Parallelization

Three important factors, one each corresponding to three parts of the **COMPASS** system, will determine the success of the **COMPASS** system: (α) effectiveness of signatures in uniquely representing program segments (matcher and watcher modules); (β) selectivity of optimization ranking procedures (database module); and (γ) efficacy of algorithms used to reconstruct code sketches from signatures (generator module). In the rest of the section, we first discuss traditional solutions to the above problems (if any), point out their drawbacks, describe alternatives we are currently working on and discuss how the proposed alternatives meet the desiderata. The following section provides only an overview of the techniques used in **COMPASS**; details are available in separate documents through the **COMPASS** website: `http://compass.cs.columbia.edu`

Traditional techniques are not suited for the sub-problems intended to be addressed by **COMPASS** for the simple reason that the traditional solutions do not scale to the size of the collaborative code base; these traditional solutions sometimes do not even scale to the size of standalone code bases because of difficulty with static analyses and code search! One may wonder how an Internet-scale code base, which is many (hundreds of thousands of) times larger than any standalone code base, can be analyzed at all.

The power to analyze these large code bases comes from two important aspects of the proposed **COMPASS** system: First, unlike traditional systems that require complete mechanization of the analysis process, **COMPASS** acknowledges the usefulness of human involvement (after all, programming is a human activity) and does not have to produce the correct advice all the time. In the rare cases when **COMPASS** may produce bad advice (e.g., advice that is apparently irrelevant to the problem at hand), such as during the early database buildup stage, the programmer is free to discard the advice and ask for any known alternatives. Secondly, and perhaps more importantly, **COMPASS** mitigates the scalability concerns by reducing the size of the code regions to be analyzed. It is widely believed that 90% of program execution time for most applications is spent in 10% of the code. **COMPASS** increases the effective scalability of traditional algorithms by focusing only on the most critical regions (that 10% of the code – or in practice, the profiled hotspots).

5.1 Program Representation Innovations

Conceptually, the **COMPASS** subsystems that use signatures to store and match parallelization solutions are similar in spirit to work on code clone detection – with two major distinctions: **COMPASS** attempts to locate similar code segments *across several code bases* with the goal of improving *performance*, while code clone tools generally aim to locate similar code segments *within a code base* to improve *maintainability*. The increased code size and the need to look across code bases, in some cases anonymously, places different stresses on the signature representation and precludes the possibility of adopting the internal representations used for detecting code clones.

§ **Related Work Analysis** Textual [8], Abstract Syntax Tree (AST) [9, 45] and token based representation [50] based methods for detecting code clones have been proposed. These encodings are effective for catching exact "cut and paste" statements in a code base, but fail when the code is semantically the same but syntactically different. For example, a for and while loop with slightly different syntactic bodies may elude

clone detection. Clearly, these representations do not have sufficient universality to be used as signatures in a collaborative optimization system. A Program Dependence Graph (PDG) [32], which can be described as a graph of program statements (nodes) and dependencies between the statements (edges), has been proposed to overcome the syntactic limitations of ASTs [33, 46]. In this representation, semantically identical code regions result in isomorphic portions, so detecting code clones reduces to the problem of colored subgraph isomorphism. This process is intractable in the general case, but solvable for small graphs (typically procedure level PDGs) in the absence of aliases. In the presence of aliases, however, PDGs may suffer from low specificity. For instance, when an ambiguous node is present in a graph, all nodes following the ambiguous node in the graph should include an edge to the ambiguous node indicating a possible dependence. As the number of ambiguous nodes increases, the number of ambiguous edges increases, consequently diminishing the power of the representation to uniquely identify code segments.

§ **Efficient COMPASS Signatures** To overcome the specificity shortcomings of PDGs, we introduce a new program abstraction called a *Segment Dependence Graph* (SDG). Like a PDG, an SDG represents dependencies between statements; but unlike the PDG, the SDG has no ambiguous edges (and thus improves specificity). In a SDG, we sidestep the problem of ambiguity of static alias analysis techniques by annotating the edges with a purely dynamic measurement of dependencies. Let us consider an example to first illustrate SDGs and then illustrate the benefits over PDGs. Consider a simple program segment with N SSA-style statements [25] $S_1, \ldots, S_N$ where S_i denotes that statement S_i executed after S_{i-1}. Also assume that the statement S_3 depends 2 times on S_2 and 3 times on S_1 during some execution of the segment. The SDG for this segment will be constructed such that the edges to S_3 from S_1 and S_2 will be annotated with the fraction of the times these dependences occur i.e., 0.4 and 0.6 respectively. To illustrate the differences from a PDG, let us now say that the statement S_2 is ambiguous, i.e., the compiler cannot statically determine if the statements following S_2 depend on S_2; to denote this the PDG for this segment will have an edge from S_2 to all statements after S_2 but the SDG will not. While in theory these run-time annotations hold only for a given input, in practice the dependency information tends to be fairly stable across inputs. We are not the first to make this observation: compiler researchers have proposed using run-time dependency information for profile-driven speculative optimizations [70] and run-time system builders have implemented some of these optimizations in production systems [26]. **COMPASS**, which does not have the same strong correctness criteria like compilers, we believe, will also stand to benefit from this kind of representation.

§ **SDG Creation** The SDG representation is used in both the watcher and matcher modules as part of the program signature. The process for creating SDGs is as follows: We expect that gurus will use a GUI-based marking tool (a sample will be provided as part of the **COMPASS** suite) to quickly mark the code regions that have been parallelized, although we will also investigate automatically matching the before and after regions using code versioning tool extensions. A lightweight program analyzer and profiler (LAP, also to be distributed as part of the **COMPASS** suite) will then transform both marked regions into SDGs. Similarly at the matcher side, LAP will create the SDGs of the most frequently executed code regions, i.e., hotspots. LAP performs fairly standard instrumented execution similar to coverage tools such as `gcov` [51].

§ **Relation to Desideratum** Since any program can be represented as a dependency graph, SDGs can be

used universally. Unlike other universal representations, SDGs have high specificity: the chance of two semantically unrelated program segments having identical SDGs and the exact same dependence probability is quite low because of the annotation based on run-time dependence. SDGs can be used for proactively fetching advice after they have been automatically created by **COMPASS**'s matcher. They are scalable since they can be created independently of the learner code base size (they are generated only for the regions of the code where most of the execution time is spent, where we expect numerous isomorphisms and rare singletons), and their structure enables (shown next) searching in a scalable manner. However, specificity and anonymity are fundamentally at odds so SDGs cannot be sent by the matcher for anonymous advice. However, the matcher can send abstract declarative queries instead of the SDGs. For example, the matcher can locally construct and analyze the SDG, then send an abstract query such as fetch "doubly nested for loops operating on three contiguous arrays with addition operations." Because such information can also be inferred from the already stored SDGs in the database, advice can still be served anonymously.

5.2 Search and Retrieval Innovations

COMPASS's knowledge database stores SDGs as $< key, value >$ pairs, in which the *key* corresponds to the unoptimized before version and the *value* is the optimized after version. Since SDGs are graphs, queries employing only a traditional relational database schema, which are typically optimized for flat text and numerical data, are unlikely to be very efficient. Further, it may be desirable to find an approximately matching SDG if an exactly matching SDG is not available in the database because minor variations in SDGs for the same program segment are to be expected. Minor variations can be a result of different environmental factors such as difference in caller-callee save conventions, register and memory allocation procedures etc., Additionally, when more than one advice exists (value) for a given SDG (key), the datastore must rank the solutions and return what the system considers to be the most useful solution(s) in order.

§ **Related Work Analysis** We do not know of any published work that has encountered or attempted to solve this problem. The closest in spirit to **COMPASS**'s proposed code search system is the string-based regular expression search service provided by the Google Labs Code Search Engine [20]. Based on the author's trial of that system (the architecture of system has not been published), the service appears to be little more than a sophisticated implementation of the unix "grep" utility over code repositories on the web, with no perceivable ranking of search results.

§ **Overview of Database Operations** Since graph isomorphism checks are computationally expensive, to enable quickly searching through hundreds of thousands of SDGs, **COMPASS** uses a series of pre-filters to first narrow the number of items that have to be searched in the database. The pre-filters can be grouped in two categories: (a) Environment filters and (b) Graph intrinsics filters. Environment filters use some features of the target machine specification (such as the instruction set architecture and cache configuration), the target compiler, the libraries available on the target machine, and the source language to reduce the search scope. The features required for environmental filtering are included by the matcher query tool as part of the segment signature. The graph intrinsic filters are filter based on the number of nodes and edges in the query SDGs.

To pick one or few candidate SDGs from the pre-filtered SDG we propose to use a new ranking method called the "code rank". The coderank metric is based on the intuition that isomorphism alone is insufficient to recommend a SDG; some times an imperfectly matching SDG but with higher perceived usefulness may be preferred, which may be the case if the perfect matching SDG is untrustworthy and intentionally seeded into the database. The coderank is computed based on three distinct characteristics: (a) the structural similarity between the key and query SDG, (b) the dependence annotation similarity between the key and query SDG and (c) the perceived importance of the value SDG corresponding to a key SDG (similar to the pagerank algorithm [60]). The structural similarity is computed as an Isomorphism score (I-score) and is defined as the fraction of subgraphs that are similar between the two graphs. The annotation similarity is computed as the Euclidean distance (E-score) between the dependence annotation between the similar edges in the two graphs. The perceived importance score (P-score) is based on a combination of factors including the improvement in performance on the target machine (transmitted when learner accepts) and the number of users using a particular advice (both gurus providing and learners accepting). An open question is to determine the weightings of the P, E and I-scores for determination of effective coderank. More details of P,E and I-scores are avaiable as seperate documents.

§ **Relation to Desideratum** The characteristics relevant to search and retrieval are Scalability, Selectivity and Anonymity. Although graph matching is computationally expensive, the relatively small size of the SDGs, combined with the pre-filtering and recent advancements in determining closeness of high-dimensional data (edge annotations) [5, 30] will enable us to construct a scalable data store. The Coderank algorithm has the potential to provide selectivity. Anonymity can be preserved since the code rank algorithm maintains only aggregate usage counts of advice.

5.3 Program Presentation Innovations

One of the core goals of **COMPASS** is to provide advice in a format that is easy for the learner to use. The advice in **COMPASS** is presented in form of a code "sketch" – an outline that shows the main control flow and data structure changes required for parallelism – closely matching the source code that the user would like to parallelize. The key to such a sketch is converting the parallelized SDGs into optimized code sequences tailored to the user's code context. The challenge is that a SDG can be translated to source code in many ways, as it does not preserve syntactic information. For example, it is easy to detect a loop from a SDG but difficult to say whether the loop is a `for` loop or a `while` loop. Similarly, it may be possible to say two arrays are being added but may not be possible to guess the names of the arrays when there are more than two choices in the context. For an effective usable sketch, a method for inferring ambiguous syntactic information is required.

§ **Related Work Analysis** Decompilers [29], Source-to-Source translators [3] and Pretty print tools [58] are commonly used to translate between compiler formats. The difficulty of translation, and the subsequent quality of the output, depends on how much high-level syntactic information is preserved in the input format. Pretty printing tools and source-to-source translators, start with input that has lot of syntatic information, and generally produce reasonable output; decompilers start with assembly format in which there is very less syn-

tactic information and produce output which is suitable for mature, patient users. The conversion tool used in **COMPASS** is different from the above since **COMPASS**'s goal is to create only a "sketch" corresponding to the SDG, and not the full source code. Further, the sketch is created from *two* input sources: an existing source file (with full syntactic information) and a parallelized SDG (with little syntactic information), whereas previous works have only one input format with either full syntactic information or none.

Our work is complementary to the work on sketching languages [74, 75]. Sketching languages are meta-languages that permit users to write incomplete programs that can be automatically completed using sophisticated static analyis. Sketch analysis tools expect the user to specify program invariants and then use the invariance information along with the program structure to create fully specified programs. **COMPASS** and sketching languages can potentially enhance each others utility. Currently, sketching schemes require the humans to prepare the sketches. **COMPASS** on the other hand has the ability to automatically generate sketches. Similarly, once **COMPASS** has a sketch, if the learner's program has invariants, a sketch compiler may be able to turn the sketch into code and thus enhance learners productivity further.

§ **Sketch Generation COMPASS** uses a multi-step procedure to create a sketch from a SDG. Our algorithm operates on: (1) the unoptimized source code and its SDG (called the program SDG) and (2) the response from the data store which contains (a) the recommended SDG (the *value* from some $< key, value >$ pair), called the advice SDG, and (b) the SDG corresponding to the advice SDG in the data store (the key from that pair), called the key SDG (which is not necessarily the same as the program SDG sent by the matcher tool). First, the program SDG is annotated with the missing syntactic information based on source code analysis (the program SDG, by design does not contain syntactic information to provide anonymity.) Then the annotated program SDG is compared to the key SDG and semantically similar statements between the two SDGs are used to deduce a mapping for variable names in the key SDG. These variable names are then used to annotate the advice SDG by a simple substitutions. When the advice SDG has been customized to the maximum possible extent based on the information available in the unoptimized source code, the next step is to convert the advice SDG into higher level source code. The control flow structures will be determined based on the structure of the graph followed by statement substitution in the control flow blocks. Finally, the sketch (with some possibly undeterminable portions) is displayed to the user as a graphical overlay.

§ **Implementation** The LAP tool provides the above functionality. In addition, a feedback agent is used to transmit information on the quality of the advice to the data store. After the sketch is generated, the feedback agent collects and transmits information on (1) the usefulness of the advice (determined subjectively by the user) (2) the number of undeterminable variable names in the sketch and (3) the performance of the program after the advice is used by the learner. The user can optionally choose not to provide feedback.

§ **Relation to Desideratum** The desideratum relevant to the sketch generator are Specificity, Scalability, Universality and Anonymity. The proposed approach can provide Specificity as it provides a sketch that is customized to the learner's source code. The approach is scalable because it creates sketches only for hot program segments instead of full programs. Further, as this approach can be applied to different languages and environments it is also universal. The approach also preserves anonymity since no information regarding the program is transmitted to the code base and since the feedback contains only summary information.

6 Metrics, Evaluation and Ongoing Efforts

The ideal evaluation for any community-based code mining system, i.e., its actual adoption and use, requires a long term retrospective view. However, measurement of some of the desiderata(Sec. §3) can bear out the effectiveness of the **COMPASS** during the life time of the system. To measure how **COMPASS** performs with regard to the specificity desideratum, we will need to measure how well the advice provided by the system relates to the source code the learner is trying to parallelize. Specificity losses may occur in **COMPASS** at two places: (a) in the data store because of weak matching metrics or (b) at the sketch generator because of poor deduction of variable names and control structures based on the code at hand. For the former, we plan to seed the database with an exact key SDG as the SDG supplied by the programmer to determine whether the data store ever fails to identify this case (unlikely). For the latter, we will manually produce a high level representation corresponding to the advice and calculate the Levinstein distance (the number of tokens that have to be changed to get from one string to another) between the generated advice and the hand-coded advice. Both measures will also be informed by feedback from users after **COMPASS** is deployed. The universality of the system can be recognized by enumerating the architecture types, language types, compilers, etc. available in the data store. Another metric for universality is the fraction of matcher queries that do not have exact system matches. We will prominently display these metrics on the **COMPASS** portal to draw users to contribute advice on missing samples. Note that **COMPASS** can provide advice even in the absence of exact system matching data, but the specificity may vary. We will monitor scalability as the rate of queries the data store can handle and the average codebase size to which system advice has been applied. **COMPASS** by design can provide anonymity if the user so desires. The measurement of theoretical limits for these desiderata is an open question.

COMPASS is currently capable of analyzing C/C++ codes only and is built around several open source tools available on the Linux platform. We use the gcov [51] coverage tool to identify frequently executed regions; use an Eclipse [37] plugin for adjusting the hotspot regions identified by gcov; leverage LLVM [47], a compiler research infrastructure, to generate signatures; and employ a simple SQL database for storing and matching the signatures. With the current infrastructure, **COMPASS** can already analyze enterprise-scale projects such as the open source javascript engine v8 [21]. **COMPASS** can identify the most profitable regions for parallelization, display these regions graphically to the user, create signatures from the code, send signatures to a database, and obtain matches when one exists. We seeded the database with a few hand-coded sample optimizations. Efforts are underway to build sketching capabilities and more sophisticated database schema. We hope to open the system for trial release to a select group of users in the last quarter of 2009. Users who would like to use the system are requested to register for an account at: `http://compass.cs.columbia.edu`

7 Conclusions

The trend towards processors with multiple cores on a chip (and implicitly the emphasis on concurrency for performance improvements) is likely to continue for the foreseeable future because of VLSI technology trends. The move to multicores has forced software developers to either parallelize code or accept significant performance slowdown (because multicores have lower clock frequency than their uniprocessor

predecessors). In this report, we have described a vision and provided implementation details for a world-wide collaborative sysetem for rapid parallelization of the code. The collaborative system, called **COMPASS** can dramatically increase programmer productivity when attempting to leverage today's and the future's computer hardware, and thus retain performance improvements in line with historical trends.

COMPASS uses an alias free representation of program segments as a basis for a code search engine that is used to locate parallelization solutions contributed by users. The results from the search engine are ranked using heuristics and the most relevant parallelization strategy is returned to the user in a custom format that can be used as a starting point for parallelization. We believe that **COMPASS** as currently designed is particularly suitable for rapid incremental parallelization of sequential codes for CMPs. Alternative approaches such as complete re-write of applications using new languages, or automatic compiler analysis are likely to be impractical for the largely sequential and diverse code bases developed over decades of computing.

To promote widespread use of **COMPASS** , we believe that **COMPASS** can be packaged as a marketplace for buying, selling and sharing program optimizations – as a "flickr for program optimizations". A potential licensing model that may be plausible for **COMPASS**-based optimizations is inspired by the recent work by Lessig on hybrid Internet economies [48], which allow for-profit and for-free software to co-exist. Lessig argues that such a mechanism (typically specified by the use of a Creative Commons license) is essential to promote legal sharing of code. Although Lessig discusses the issues from a point of view of art, the same model can also be applied to code, as our proposed model is, in principle, to remix different parts of existing code to create a new and improved form of existing code segments.

COMPASS is not just a reactive solution to fundamental changes in computer architecture; it provides an infrastructure that can proactively influence and aid in the design of new computer systems. In the steady-state, **COMPASS** can be data mined to determine the most frequently requested optimizations – which can be utilized by computer architects to create new hardware extension units or by compiler writers to generate targeted compiler analyses that speedup these frequently requested (executed) regions.

Finally, we envision that one day in the not so distant future, that programming could be as simple as a typing a series of queries on a special search engine. Imagine the productivity improvements possible if a complex and diverse (and profitable) application such as game for a highly constrained mobile device could be created quickly by typing a terms such as "real time ray tracer", "neural network", "high score Internet enabled DB", "car model selection code", "player physics logic" etc., and sketching out some glue logic. As outlined in §5.3, a combination of advances in sketching systems, collaborative code mining systems such as **COMPASS** and query interfaces has the potential to bring us closer to this programming utopia.

8 Acknowledgements

The authors would like to thank Sal Stolfo for discussions and Al Aho for a discussion and for suggesting the use of Levinstein distance for measuring specificity of the sketcher. We would like to thank the Nipun Arora and Bing Wu for work on the **COMPASS** pilot project and Ravindra Babu Ganapathi for help with the hand-optimized code samples. The Programming Systems Laboratory is funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1. The Computer Architecture Laboratory is funded in part by AFRL FA8650-08-C-7851.

References

[1] Aart J.C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, May 2004. (Cited on page 6.)

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000. (Cited on pages 2 and 4.)

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cited on page 13.)

[4] Shameem Akthar. *Multi-Core Programming: Increasing Performance through Software Multithreading*. Intel Press, USA, 2006. (Cited on page 6.)

[5] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008. (Cited on page 13.)

[6] Abraham Arevalo, Ricardo M. Matinata, Maharaja (Raj) Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Corporation, USA, 2008. (Cited on page 6.)

[7] Krste Asanovic, Ras Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John D. Kubiatowicz, Edward A. Lee, Nelson Morgan, George Necula, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008. (Cited on page 2.)

[8] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society. (Cited on page 10.)

[9] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on page 10.)

[10] Jon Bentley. *Programming Pearls (2nd Edition)*. Addison-Wesley Professional, September 1999. (Cited on page 6.)

[11] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1):27–59, March 2005. (Cited on page 6.)

[12] Walt Brainerd. Fortran 77. *Commun. ACM*, 21(10):806–820, 1978. (Cited on page 4.)

[13] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM. (Cited on pages 2 and 4.)

[14] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. (Cited on page 5.)

[15] Marcelo H. Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, pages 13–24, 2000. (Cited on page 2.)

[16] D. Clark. Openmp: a parallel standard for the masses. *Concurrency, IEEE*, 6(1):10–12, Jan-Mar 1998. (Cited on pages 4 and 9.)

[17] J. Cocke and D. Slotnick. The use of parallelism in numerical calculations. Research Memorandum RC-55, IBM, 1958. (Cited on page 4.)

[18] Douglas Comer and Larry Peterson. *Network Systems Design Using Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003. (Cited on page 6.)

[19] Keith Cooper. Compilation technology for tailoring applications to multiple architectures. `www.darpa.mil/IPTO/personnel/docs/Attachment_04.ppt`, November 2007. (Cited on page 4.)

[20] Google Corporation. The google code search engine. `http://http://www.google.com/codesearch`. (Cited on page 12.)

[21] Google Corporation. V8 javascript engine. `http://code.google.com/p/v8/`. (Cited on page 15.)

[22] NVIDIA Corporation. Ageia physx product brief. `http://www.nvidia.com/object/physx_accelerator.html`. (Cited on page 6.)

[23] Isom L. Crawford and Kevin R. Wadleigh. *Software Optimization for High Performance Computers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. (Cited on pages 5 and 6.)

[24] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998. (Cited on pages 6 and 9.)

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. (Cited on page 11.)

[26] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 11.)

[27] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965. (Cited on page 4.)

[28] Max Domeika. *Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture*. Addison-Wesley Professional, April 2008. (Cited on page 6.)

[29] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 13.)

[30] Kave Eshghi and Shyamsundar Rajaram. Locality sensitive hash functions based on concomitant rank order statistics. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 221–229, New York, NY, USA, 2008. ACM. (Cited on page 13.)

[31] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, 2007. (Cited on page 2.)

[32] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. (Cited on page 11.)

[33] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM. (Cited on page 11.)

[34] Rajat P. Garg and Illya Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. (Cited on page 6.)

[35] Lal George and Matthias Blume. Taming the ixp network processor. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM. (Cited on page 2.)

[36] José González and Antonio González. Limits of instruction level parallelism with data value speculation. In *VECPAR '98: Selected Papers and Invited Talks from the Third International Conference on Vector and Parallel Processing*, pages 452–465, London, UK, 1999. Springer-Verlag. (Cited on page 4.)

[37] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Syst. J.*, 44(2):289–299, 2005. (Cited on page 15.)

[38] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM. (Cited on pages 2 and 4.)

[39] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. (Cited on pages 4 and 6.)

[40] http://forums.amd.com/devblog/categories.cfm?catid=209. Hard-core Software Optimization. (Cited on page 5.)

[41] http://software.intel.com/en-us/articles/pentium/all/1. . (Cited on page 5.)

[42] http://www.khronos.org/registry/cl/. OPenCL specification. (Cited on page 2.)

[43] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004. (Cited on page 4.)

[44] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967. (Cited on page 4.)

[45] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 10.)

[46] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 11.)

[47] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. (Cited on page 15.)

[48] Lawrence Lessig. *Remix: Making Art and Commerce Thrive in the Hybrid Economy*. Penguin Press, 2008. (Cited on page 16.)

[49] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, 2005. (Cited on page 2.)

[50] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 10.)

[51] GNU GPL License. Gcov: Gnu coverage tool. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. (Cited on pages 11 and 15.)

[52] LLNL. https://computing.llnl.gov/tutorials/mpi/. (Cited on page 5.)

[53] LLNL. https://computing.llnl.gov/tutorials/pthreads/. (Cited on page 5.)

[54] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004. (Cited on page 5.)

[55] Wen mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, New York, NY, USA, 2007. ACM. (Cited on page 2.)

[56] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001. (Cited on page 2.)

[57] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008. (Cited on pages 2, 6 and 9.)

[58] Dereck C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980. (Cited on page 13.)

[59] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 2.)

[60] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998. (Cited on page 13.)

[61] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005. (Cited on page 6.)

[62] Chuck Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23(4):298–298, 2008. (Cited on page 2.)

[63] Graham D. Price, John Giacomoni, and Manish Vachharajani. Visualizing potential parallelism in sequential programs. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 82–90, New York, NY, USA, 2008. ACM. (Cited on page 4.)

[64] Daniel Reed. The law of the pack. *Harvard Business Review*, pages 2–3, 2001. (Cited on page 3.)

[65] Shane Ryoo. *Program Optimization Strategies for Data-Parallel Many-Core Processors*. PhD thesis, University of Illinois at Urbana-Champaign, April 2008. (Cited on page 6.)

[66] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp and dlp with the polymorphous trips architecture. In *ISCA*, pages 422–433, 2003. (Cited on page 2.)

[67] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, December 2006. (Cited on page 4.)

[68] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008. (Cited on page 2.)

[69] Simha Sethumadhavan. *Scalable Hardware Memory Disambiguation*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, December 2007. (Cited on page 4.)

[70] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. *SIGPLAN Not.*, 41(11):416–425, 2006. (Cited on page 11.)

[71] Simcha Gochman and Avi Mendelson and Alon Naveh and Efraim Rotem. Introduction to the Intel Core Duo Processor Architecture. *Intel Technology Journal*, 10(2), May 2006. (Cited on page 4.)

[72] Simha Sethumadhavan. COMS E6998-1: Advanced Computer Architecture Parallel Systems and Programming. `http://www.cs.columbia.edu/~simha/teaching/6998_fa08/`, 2008. (Cited on page 5.)

[73] Aaron Smith, Ramadass Nagarajan, Karthikeyan Sankaralingam, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. Dataflow predication. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–102, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 2.)

[74] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 167–178, New York, NY, USA, 2007. ACM. (Cited on page 14.)

[75] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Notices*, 41(11):404–415, 2006. (Cited on page 14.)

[76] E. Stewart. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004. (Cited on pages 6 and 9.)

[77] Sun Microsystems Inc. Developing and Tuning Applications on the UltraSPARC T1 Chip Multithreading Systems. October 2007. (Cited on page 6.)

[78] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005. (Cited on page 1.)

[79] Scott Vetter, Stephen Behling, Peter Farrell, Holger Holthoff, Frank O'Connell, and Will Weir. *The POWER4 Processor Introduction and Tuning Guide*. IBM Corporation, USA, 2001. (Cited on page 6.)

[80] Luis von Ahn, Ruoran Liu, and Manuel Blum. Peekaboom: a game for locating objects in images. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 55–64, New York, NY, USA, 2006. ACM. (Cited on page 3.)